JOT - A Specification for an Ink Storage and Interchange Format

Version 1.0

This document represents the joint work of Slate, Lotus, GO, Microsoft, Apple, General Magic, and others.

since 14/Mar/1996

Please address correspondence or inquiries about this document to:

Slate Technical Support
Slate Corporation
15035 N. 73rd Street
Scottsdale, Arizona 85260
(602) 991-6844
(602) 443-3606 Fax
75300,3236 on CompuServe

Jot Specification Overview

So much of the information we process and communicate every day takes the form of handwritten notes and drawings. The value of using a computer to manage, organize, and communicate this type of human expression is obvious. With a pen computer, a user can pick up an electronic pen and write or draw directly on the screen. What appears is electronic ink which can take the form of handwritten notes, sketches, signatures, and fill-out forms.

Until now no standard format existed for how ink should be stored and represented electronically. This lack of definition severely limits how this information can be shared between users, applications, and machines. Responding to the need to share ink-based information, technical representatives from Slate, Apple, General Magic, GO, Lotus and Microsoft met under the leadership of Slate to address the issues of a common ink format. The result of this effort is the Jot specification.

Jot defines a format for the storage and interchange of electronic ink between software applications. Jot is a platform and application independent specification. It provides a simple, convenient, and open format that maintains complete likeness with the original ink as it was drawn. Maintaining fidelity to the original is what ensures ink can be shared and understood across many environments. Typically, users will share ink by cutting and pasting it between applications, embedding and storing it in documents or databases, and importing and exporting it to and from applications.

Jot is similar in intent to existing computer format specifications such as TIFF, GIF, PCX. However, Jot specifies the richness and depth of attributes required for ink that are not provided by existing data formats.

The Benefits of Jot

The benefits of the Jot specification fall mainly into three areas:

1. The ability to share ink-based information independent of the platform on which it was created. Jot is platform independent, which means that ink created on one device can be shared with another, even if the destination device does not support the pen. Ink stored in the Jot format can be displayed on any system capable of graphics.
2. The ability to maintain full likeness to the original ink even after it is compressed. Jot allows the receiving application to preserve all of the original ink attributes. Ink stored in this way can be manipulated at a later time as if it were just entered. For example, ink can be translated by a recognizer, printed on a high-resolution printer, or scaled without loss of precision relative to the original device on which it was created.
3. The ability to be open to change and still maintain forward and back compatibility across versions. No doubt the Jot specification will evolve and change over time. Jot specifies a self-describing record-based structure to define ink data. Applications can selectively act upon any or all of the attributes recorded for a specific piece of ink. Application developers and users won't have to continually upgrade applications to keep pace with Jot changes.

Uses of Jot

Jot enables ink to be used across a very broad range of applications and devices. With a standard interchange format, a number of scenarios are possible. Here are a few examples of ink sharing. Of course, many more applications will arise as Jot is implemented on diverse platforms.

1. Sharing signatures and annotations between mobile, pen-based computers and a central database. Very often, companies need to capture signatures and annotations as part of data intensive, forms-based applications. In most cases, field collected information must be communicated back to a corporate database for storage and future analysis. Today, non-pen systems have no way to handle ink, except as a bit-map or equivalent format. With Jot, the application can implement ink on the portable system and as part of the central database system. Then signatures, sketches, notes and more can be exchanged between the two systems and throughout an organization.
2. Sharing electronic mail between handheld devices and desktop systems. There is a need to access e-mail when you are away from the office. Using a portable pen-based device is an excellent way to do this. Not only can users view traditional text-based messages, they can mark up existing messages or create new ones using the pen. Messages can contain handwritten replies, sketches, annotations and doodles. In this case, the underlying operating system may not be the same on the sending or receiving systems. And the desktop system doesn't require a pen interface at all. Jot supports transfer of handwritten e-mail between incompatible systems.
3. Taking and sharing notes throughout an organization. Whether taking notes individually in a meeting or working with a group, note taking is a natural application for ink. Note taking applications written to Jot will be able to organize and share this kind of free-form information among a broad range of devices from handheld- to white board-size systems.

Audience for Jot

The Jot specification is useful by a broad range of system specifiers, designers, and developers, including:

1. Application and system software developers. The specification is for software writers who want to implement ink as a standard and compatible data type. It includes enough detail and sample code to ensure compatibility across a broad range of applications and platforms. The specification is written primarily in "C" language constructs to speed implementation and to ease understanding. The specification can, however, be implemented in any language.

2. Digitizer and hardware designers. Digitizer and hardware designers will benefit from understanding Jot because it provides a model for ink as it will be used by applications. They will be able to design better and more efficient systems to manage, store, and display ink data.
3. Product managers. People involved in product management and feature specifications for software products will want to be familiar with the specification at a high level in order to make informed decisions about how much support of the specification a particular application should provide.

Goals for Jot

The overall objective of Jot is to define ink in very detailed terms and then leave it up to the software application to recognize any or all of the attributes for ink. Jot includes more than just the simple x,y coordinates specifying where ink was drawn on a page. It includes detailed information about the pen tip used, the color of the ink, the sampling rate of the digitizer, the angles used to write, and more.

The ink storage format has a number of goals:

Simple.

Typical operations on the ink data are easy. For example, if an application only needs the stroke and bounding information, other properties present in the record are just ignored by the application.

Compact.

The storage format is optimized for reduced space. Optional information such as time stamps or color specifications occupy space only when they are present. Specifications that apply to many strokes, such as line width or color, are represented just once.

Compression.

Jot facilitates reducing the amount of storage required for ink by including both a lossless compression scheme for stroke information and the ability to optionally reduce the amount of information retained for a particular piece of ink.

Inclusive.

The format can store every conceivable property of ink data known at this time.

Expandable and Compatible.

The format is expandable, so as developers discover new properties for ink, they can be added without changing the behavior of existing application programs working with an older version of the format. New features can be ignored by applications reading older versions of the format. Likewise, new application programs can handle previous versions of the format without special work.

The format is not designed to support memory-intensive manipulation of the ink data such as deleting strokes, changing line widths, and so on. A format supporting these types of manipulations would be at odds with the above goals. However, all of the information needed to perform these operations is present in the data format. So an application could augment this format to provide manipulation of the ink data. In practice, the external storage format will probably not be the same as the internal format used by an application. The exception might be for those applications that do not support ink editing or transformation and are only storing ink data as part of a document.

The specification is a record-based format. This means that each piece of information is stored in a record structure and contains enough self-describing information so an application unfamiliar or unconcerned with a particular record item can skip it.

Supported Ink Properties

One question often asked is "Why don't you just store ink as a bit-map?" Bit-maps describe the relative position on a page of a series of dots or bits that make up a particular image. Ink data requires much more information than just the position of dots on the page. For example to perform certain ink operations such as translating hand printed input to ASCII characters, an application needs to know how ink was created as well as what was drawn. With ink you need to maintain complete likeness to the original. To maintain this

fidelity, ink is described by a wide variety of properties. Applications can choose to recognize or ignore properties as required. These properties are stored in an application-specific record. Specified properties include:

- Multiple strokes of ink combined into single objects
- Bounds
- Scale
- Offset
- Numbered groupings of strokes
- Color, including opacity
- Pen tips
- Timing information
- Height of the pen over the digitizer
- Stylus tip force
- Buttons on the stylus
- X and Y angle of the stylus

In addition to these properties, there are reserved record types for future expansion.

The Future of Jot

The intent of the Jot specification is similar to other standards widely used in the software industry. It contains language permitting developers to incorporate and otherwise use any portion of the specification without hindrance. Slate Corporation will collect comments and suggestions for future versions of Jot. The ad hoc committee of Apple, General Magic, GO, Lotus, Microsoft, and Slate will meet regularly to review and ratify any changes to the specification.

Anyone can receive a copy of the Jot specification by contacting the Software Publishers Association (SPA) at (202) 452-1600 extension 336.

```
/*------------------------------------------------------------
** File: inkstore.h
**
** Copyright 1993, Slate Corporation, All Rights Reserved.
**
** This document is part of the Jot specification for the storage and
** interchange of electronic ink data. This specification is the joint work
** of representatives of Slate Corporation, Lotus Development Corporation,
** GO, Microsoft, Apple, General Magic, and others.
**
** This document and the accompanying code samples on disk comprise Version
** 1.0 of the Jot specification for the storage and interchange of electronic
** ink data. Permission is granted to incorporate and otherwise use any
** portion of the specification. You may make copies of the specification
** for distribution to others, provided you include the notice "Copyright
** 1993, Slate Corporation. All Rights Reserved" on both the document and
```

```
** the disk label.  You may not modify this specification without written
** permission from Slate Corporation.
**
** The specification is provided "as is" without warranty of any kind.  Slate
** further disclaims all implied warranties of merchantability or of fitness
** for a particular purpose.  The entire risk arising out of the use or
** performance of the specification remains with you.
**
**--------------------------------------------------------------------
**
** This is the main body of definitions for the ink storage specification.
** See reference section 1.0 for revision history.
**
**--------------------------------------------------------------------*/

#ifndef INKSTORE_INCLUDED
#define INKSTORE_INCLUDED


/*************************/
/* REFERENCE SECTION 0.0 */
/*************************/

/*--------------------------------------------------------------------
** "Rationale for the ink specification"
**
** This document defines a storage and interchange format for embedded ink
** data.  The format is device- and platform-independent.  The goal is to
** provide application programs on the same and different platforms and
** operating systems a way to store and exchange ink data.  Thus a PenPoint
** user might scribble a note and send the untranslated ink as part of an
** e-mail message to a colleague's pen computer running Windows for Pen
** Computing using Magic Mail.
**
** This specification is for a publicly-defined, external format for
** electronic ink data interchange, and neither assumes nor dictates the
** nature of how the application deals with ink data internally.  The format
```

** is not intended to be the "internal" ink format of an application, though
** there is no reason why it could not serve such a purpose.
**
** The scope and goals of this format design are limited to the represent-
** ation of electronic ink data embedded in some other electronic document,
** not to the larger document itself (such as an e-mail or enhanced word-
** processing data file).
**
** The approach taken is to capture the complete user input for the
** electronic ink, including not just X/Y coordinates, but also a large set
** of current drawing attributes such as nib type and ink color. This
** differs from other possible approaches, such as those based on certain
** recognition models for handwritten text, which require decomposing the
** handwritten ink data first into a set of pre-defined approximation curves
** or sub-strokes, and then storing a list of encodings of these sub-strokes.
** In other words, Jot preserves all information about the original input as
** opposed attempting any sort of abstract characterization of the input.
**
** The storage format has a number of properties:
**
** * Simple. Typical operations on the ink data are easy. If you only wish
** to read stroke coordinates and bounding information from the data,
** complex information that might be present will not hinder the process.
** Likewise, it is easy to write out just simple information. The
** complex information is all optional.
**
** * Compact. The storage format is intended to be as compact as possible
** without sacrificing simplicity or fidelity. Optional information such
** as time stamps or color specifications occupy space only when they are
** present. Specifications that apply to many strokes (such as line width
** or color) are represented just once.
**
** * Compression. The stroke information that describes the ink can
** optionally be represented in a compressed format. Compression
** techniques include both compression and reduction of the ink data.
**
** * Inclusive. The format is capable of storing every property of ink

```
** conceivable as of today.
**
** * Expandable and Compatible. The format is expandable, so as developers
** discover new information that should be recorded in an ink storage
** format, these new features can be added without changing the behavior of
** existing application programs working with an older version of the
** format. In general, new features can generally be ignored by
** applications reading older versions of the format. Likewise, new
** application programs can handle previous versions of the format without
** special work.
**
** The format is not designed to easily support lots of in-memory
** manipulation of the ink data, such as deleting strokes, changing line
** widths, and so on. A format supporting these types of manipulations would
** be at odds with the above goals. All the information needed to perform
** these manipulations is present in this data format, so an application
** might augment this format to facilitate manipulation of the ink data.
**
** Applications are likely to use some other format internally for real-time
** ink manipulation. Many operating environments provide some internal means
** for storing and manipulating ink data, the details of which may be hidden
** to some extent from the application designer. Many such real-time data
** structures store fewer types of and/or less information (such as not
** preserving information about the tablet point data rate) than are covered
** in this definition.
**
**-------------------------------------------------------------------------*/


/*************************/
/* REFERENCE SECTION 1.0 */
/*************************/

/*-------------------------------------------------------------------------
** Jot Ink Specification
** -------------------
**
```

```
** Revision History:
**
** March    16, 1992 - First public draft.
** July     13, 1992 - Major rewrite to put data into a series of records.
** July     19, 1992 - Inclusion of ink-compacting definitions.
** July     30, 1992 - Change of target rect to offset.
** December 28, 1992 - Changes incorporated from August 1992 review meeting.
** February 12, 1993 - Incremental fixes due to coding experience.
** March    13, 1993 - Revised definition of a "group".
** April    12, 1993 - Release of version 0.99 of the specification. Moved
**                     reference sections 28 and 29 to a separate file called
**                     sample.h
** May      01, 1993 - Release of version 1.00 of the specification.
**                     Changed INK_OFFSET_RECORD units from twips to pen
**                     units for consistency and ease of implementation.
**                     Fixed a typo in reference section 26.0 in the diagram.
**                     The text accompanying the diagram was correct.
**                     Fixed a typo in reference section 27.0. The old text
**                     "delta-X == 0 or 1" was replaced with the correct text
**                     "delta-X == 2". The accompanying diagram was correct.
**                     Removed all sizeof() constructs and replaced with
**                     appropriate #defines to reduce compiler dependencies.
**                     Tagged all struct definitions with tag_ prefix.
**                     Added comments and reordered some existing comments.
** May      17, 1993 - Added a few more _SIZE #defines, clarified reserved
**                     values.
**
**
** GENERAL NOTES
** -------------
**
**
** Record Structure
** ----------------
**
** If not otherwise specified, all words are stored in Intel order: low-order
** word first, then high-order word, and inside of a word, low-order byte,
```

```
** then high-order byte. For example, a 32 bit quantity 0x12345678 would be
** written to the file as 0x78 0x56 0x34 0x12. The notable exception is the
** storage of point data in "standard compression" format. Sign bits are
** used to indicate item types, so the bytes are stored high-order to low-
** order (exactly opposite). See the sample code and reference section 23.0
** for more information on the compressed format. Uncompressed data is
** written in Intel order.
**
** All structures are packed for the purposes of writing to a stream.
**
** Signed integer values are two's-complement. Rectangles are stored
** x,y,w,h.
**
** These definitions are intended to insulate the sample ink compaction and
** storage code from any possible variation in item alignment or structure
** packing across architectures. The only possible area of portability
** concern lies in the use of unions in colors (see 11.0) and pen tips (see
** 14.0).
**
** Any use of units of mass to denote units of force ("grams of force"), or
** similar common misuses of physical units, are noted here with an apology
** to any purists, and should be interpreted in the common way by assuming
** one standard gravity.
**
** Record Sequence
** ---------------
**
** In this document, one piece of ink data is called an ink bundle.
** Typically this might correspond to the strokes that make up the ink from
** the time when the pen touches down until the user finishes writing
** (usually determined by a timeout or the pen leaving proximity). Thus an
** ink bundle usually contains many ink strokes, and the strokes do not have
** to describe a continuous line of ink.
**
** As stated in reference section 5.0, all data conforming to this
** specification appears as a stream of ink bundles each of which must begin
** with an INK_BUNDLE_RECORD and end with an INK_END_RECORD. There may be
```

```
** more than one INK_BUNDLE_RECORD/INK_END_RECORD pair in a given stream.
** A record stream might look something like this:
**
** INK_BUNDLE_RECORD        required    // for bundle number one
** INK_SCALE_RECORD         optional    // sets the scale for rendering
** INK_OFFSET_RECORD        optional    // sets the offset for rendering
** INK_COLOR_RECORD         optional    // sets the color for rendering
** INK_START_TIME_RECORD    optional    // sets the relative start time
** INK_PENTIP_RECORD        optional    // sets the pentip for rendering
** INK_GROUP_RECORD         optional    // tags the following PENDATA
** INK_PENDATA_RECORD       recommended // actual points
** INK_GROUP_RECORD         optional    // tags the following PENDATA
** INK_PENDATA_RECORD       recommended // actual points
** INK_PENDATA_RECORD       recommended // more points in same group
** INK_SCALE_RESET_RECORD   optional    // resets to default scaling/offset
** INK_PENDATA_RECORD       recommended // actual points
** INK_END_TIME_RECORD      optional    // relative time inking ended
** INK_END_RECORD           required    // end of bundle number one
**
** It is perfectly reasonable to write out only the following (though doing
** so will cause the ink to be rendered in a completely default manner --
** black hairline width at 1:1 scaling with offset 0):
**
** INK_BUNDLE_RECORD
** INK_PENDATA_RECORD
** INK_END_RECORD
**
**
** Specification Revisions
** -----------------------
**
** Future enhancements to this specification may modify certain record types.
** It is guaranteed that any record modified in a subsequent revision of the
** specification will be a strict superset of that record's definition in any
** previous revision of the specification.  That is, modified record types
** will only be lengthened, not shortened.  If a particular record type must
** be extended such that it would not be a superset of the original, a new
```

```
** record type would be added to cover that particular extension.
**
** This extension strategy has two important ramifications:
**
** 1) A reading application should *ALWAYS* use the size of a record as
**    recorded in the record structure itself (i.e., the recordLength field
**    of the INK_RECORD_HEADERx structure) rather than the sizeof() or any
**    other size determined at compile time to determine how may bytes to
**    read as the data structures are parsed. This is due to the fact that
**    a record may grow in a future revision of the standard. The only
**    exception to this rule is the INK_BUNDLE_RECORD which contains a
**    version number that will be modified with each change to that record.
**    If an INK_BUNDLE_RECORD is encountered and its version matches the
**    version used at compile time, the size of the record should exactly
**    match the #define of inkRecordBundleSize.
**
** 2) Any particular record may be read into a target data structure up to
**    the size of the target data structure and the rest may be ignored.
**    This is due to the 'strict superset' rule which means that any
**    extension of any record type must leave the meaning, content, and size
**    of any existing fields as is. So, for example, if an INK_SCALE_RECORD
**    was modified by adding 2 bytes, the reading application can safely read
**    the data into the INK_SCALE_RECORD known at compile time and throw
**    away the extra two bytes: the header, x, and y will be in the same
**    place and will have the same meaning.
**
**
** Files of Ink
** ------------
**
** It is a recommended practice on DOS and UNIX style file systems to use the
** extension ".JOT" for files consisting solely of ink recorded according to
** this specification. The specification is designed such that ink data can
** be embedded inside any file format and if such a file contains more than
** strictly ink data, it should not use the .JOT extension.
**
**-------------------------------------------------------------------------*/
```

```
/*************************/
/* REFERENCE SECTION 2.0 */
/*************************/

/*-----------------------------------------------------------
** Definitions used in this header.
**
** These definitions must be defined appropriately to the target environment
** and compiler.
**
** For example, on some compilers for environments using segmented addressing
** and 64K segments sizes, the correct definition of FAR would be "_huge",
** rather than "_far", because the objects pointed to may be larger than 64K.
**
** In particular, check the definitions of FAR, U32, and S32 for
** compatibility with your compiler, environment, and memory model.
**
**------------------------------------------------------------*/

#ifndef FAR
#define FAR
#endif

/* useful constants */

#define flag0       (0x0001)
#define flag1       (0x0002)
#define flag2       (0x0004)
#define flag3       (0x0008)
#define flag4       (0x0010)
#define flag5       (0x0020)
#define flag6       (0x0040)
#define flag7       (0x0080)
```

```
#define flag8       (0x0100)
#define flag9       (0x0200)
#define flag10      (0x0400)
#define flag11      (0x0800)
#define flag12      (0x1000)
#define flag13      (0x2000)
#define flag14      (0x4000)
#define flag15      (0x8000)

#define flag16      (0x00010000L)
#define flag17      (0x00020000L)
#define flag18      (0x00040000L)
#define flag19      (0x00080000L)
#define flag20      (0x00100000L)
#define flag21      (0x00200000L)
#define flag22      (0x00400000L)
#define flag23      (0x00800000L)
#define flag24      (0x01000000L)
#define flag25      (0x02000000L)
#define flag26      (0x04000000L)
#define flag27      (0x08000000L)
#define flag28      (0x10000000L)
#define flag29      (0x20000000L)
#define flag30      (0x40000000L)
#define flag31      (0x80000000L)

#define TRUE   1
#define FALSE  0

/* void pointers */
typedef void        FAR *P_UNKNOWN;
typedef P_UNKNOWN   FAR *PP_UNKNOWN;

#define pNull       ((P_UNKNOWN)0)

/* Unsigned integers */
typedef unsigned char   U8,  FAR *P_U8;
```

```
#define U8_SIZE        1
typedef unsigned short  U16, FAR *P_U16;
#define U16_SIZE       2
typedef unsigned long   U32, FAR *P_U32;
#define U32_SIZE       4

/* Signed integers */
typedef signed char     S8,  FAR *P_S8;
#define S8_SIZE        1
typedef signed short    S16, FAR *P_S16;
#define S16_SIZE       2
typedef signed long     S32, FAR *P_S32;
#define S32_SIZE       4

/* geometry structures */
typedef struct tag_XY32 {
   S32    x;
   S32    y;
} XY32, FAR *P_XY32;
#define XY32_SIZE (S32_SIZE+S32_SIZE)

typedef struct tag_XY16 {
   S16    x;
   S16    y;
} XY16, FAR *P_XY16;

// Note:
// Angles from vertical can exceed +-90 degrees: in this case, the "back" end
// of the stylus is nearer the tablet surface than the "front" end.

// Note:
// Standard compaction will normally store angles in nibbles, or single
// bytes, rather than in four-byte records.

typedef struct tag_ANGLE16 {
   S16    theta;  // "X" angle of the stylus, degrees from vertical,
                  // increasing in the positive "X" direction.
```

```
   S16     phi;   // "Y" angle of the stylus.
} ANGLE16, FAR *P_ANGLE16;
#define ANGLE16_SIZE (S16_SIZE+S16_SIZE)

typedef struct tag_SIZE32 {
   S32     w;
   S32     h;
} SIZE32, FAR *P_SIZE32;
#define SIZE32_SIZE (S32_SIZE+S32_SIZE)

typedef struct tag_SIZE16 {
   S16     w;
   S16     h;
} SIZE16, FAR *P_SIZE16;
#define SIZE16_SIZE (S16_SIZE+S16_SIZE)

// Note:
// A rect where xmin==xmax and ymin==ymax has a size of zero:
// size.w and size.h are both zero.

typedef struct tag_RECT32 {
   XY32    origin;
   SIZE32  size;
} RECT32, FAR *P_RECT32;
#define RECT32_SIZE (XY32_SIZE+SIZE32_SIZE)

typedef U32 FIXED_FRACTION; // fixed point value, unity = 0x00010000
#define FIXED_FRACTION_SIZE U32_SIZE

#define INK_UNITY_SCALE ((U32) 0x00010000L)


/*************************/
/* REFERENCE SECTION 3.0 */
/*************************/


/*----------------------------------------------------------------
```

```
** A block of ink data is called an ink bundle. Each ink bundle consists of
** a series of n records. Each record has a common header that indicates the
** record type and the record length. An ink bundle always starts with an
** INK_BUNDLE_RECORD and always ends with an INK_END_RECORD.
**
** Any records of unknown type can be skipped by simply reading the length of
** the record.
**
** Note:
** Within an ink bundle, time increases. This implies a drawing order of
** back-to-front. Between adjacent sequential bundles, the implicit drawing
** is also back-to-front.
**
** A number of record types are defined. The most common is the
** inkRecordPenData which contains the actual pen data. Other records are
** mostly attributes of the pen data and are optional. They will, in
** general, only be present when a given attribute changes to something
** different than the default value for that attribute.
**
** In order to have the most compact format and also allow large records,
** several different record headers are defined, each with a different
** length.
**
** The top two bits of the record type indicate what kind of record length
** follows:
**
** The record length can be:
**
** -  non-existent (the entire record consists of just the recordType)
** -  An 8 bit length (one byte) for records up to 255 bytes
** -  A 16 bit length (two bytes) for records up to 64k
** -  A 32 bit length (four bytes) for really big records
**
**---------------------------------------------------------------------*/

#define inkRecordNoLength        0      // no length, just recordType
#define inkRecordLength8         flag14 // 8 bit length
```

```
#define inkRecordLength16    flag15           // 16 bit length
#define inkRecordLength32   (flag15 | flag14)  // 32 bit length

// useful defines for isolating or clearing the length type bits
#define inkRecordLengthMask (flag15 | flag14)  // mask for length bits
#define inkRecordLengthClearMask (~inkRecordLengthMask)

// some useful macros for declaring the various types of record types
#define MakeRec0(recType)   (recType | inkRecordNoLength)  // no rec length
#define MakeRec8(recType)   (recType | inkRecordLength8)   // 8 bit length
#define MakeRec16(recType)  (recType | inkRecordLength16)  // 16 bit length
#define MakeRec32(recType)  (recType | inkRecordLength32)  // 32 bit length

typedef U16 INK_RECORD_TYPE, FAR *P_INK_RECORD_TYPE;
#define INK_RECORD_TYPE_SIZE U16_SIZE

#define inkRecordHeaderLength(record_type) \
  ( (((record_type) & inkRecordLength32) == inkRecordNoLength) ?\
          INK_RECORD_TYPE_SIZE                  \
   : (((record_type) & inkRecordLength32) == inkRecordLength8)  ?\
          INK_RECORD_TYPE_SIZE+U8_SIZE          \
   : (((record_type) & inkRecordLength32) == inkRecordLength16) ?\
          INK_RECORD_TYPE_SIZE+U16_SIZE         \
   :      INK_RECORD_TYPE_SIZE+U32_SIZE         \
  )

// Note: most compilers will not generate code for the above macro but will
// determine the proper value at compile time.


/*************************/
/* REFERENCE SECTION 4.0 */
/*************************/

/*-----------------------------------------------------------------
** These are all the currently defined record types.  The macro MakeRecX()
** encodes the right bits in with the record Id to define its recordLength.
```

```
**
** For simplicity, recType values may not be repeated for different
** INK_RECORD_TYPEs. Use of a record type defined as MakeRec32(63) thus
** forbids the use of a record type defined as MakeRec16(63), MakeRec8(63),
** or MakeRec0(63).
**
** Record type 63 is reserved explicitly for possible future extension beyond
** 63 record types.
**
**-----------------------------------------------------------------------*/

#define inkRecordEnd            MakeRec0( 0) // end of bundle
#define inkRecordBundle         MakeRec8( 1)
#define inkRecordPenData        MakeRec32( 2)
#define inkRecordScale          MakeRec8( 3)
#define inkRecordScaleReset     MakeRec0( 4)
#define inkRecordColor          MakeRec8( 5)
#define inkRecordTip            MakeRec8( 6)
#define inkRecordGroup          MakeRec8( 7)
#define inkRecordOffset         MakeRec8( 8)
#define inkRecordStartTime      MakeRec8( 9)
#define inkRecordEndTime        MakeRec8( 10)
#define inkRecordPointsPerSecond MakeRec8( 11)
#define inkRecordUnitsPerZ      MakeRec8( 12)
#define inkRecordUnitsPerForce  MakeRec8( 13)

// Record types 14 .. 61 are reserved for future definition.

#define inkRecordApp            MakeRec32(62) // application-specific records
#define inkRecordExt            MakeRec32(63) // reserved for extension

// Every record starts with a header that contains the recordType and the
// recordLength. The recordType indicates the type of data here. The
// recordLength indicates the total length of all the data for the record
// (including the size of the header).

/* no recordLength */
```

```
typedef struct tag_INK_RECORD_HEADER0 {
   INK_RECORD_TYPE     recordType;
} INK_RECORD_HEADER0, FAR *P_INK_RECORD_HEADER0;

/* 8 bit recordLength */
typedef struct tag_INK_RECORD_HEADER8 {
   INK_RECORD_TYPE     recordType;
   U8                  recordLength;
} INK_RECORD_HEADER8, FAR *P_INK_RECORD_HEADER8;

/* 16 bit recordLength */
typedef struct tag_INK_RECORD_HEADER16 {
   INK_RECORD_TYPE     recordType;
   U16                 recordLength;
} INK_RECORD_HEADER16, FAR *P_INK_RECORD_HEADER16;

/* 32 bit recordLength */
typedef struct tag_INK_RECORD_HEADER32 {
   INK_RECORD_TYPE     recordType;
   U32                 recordLength;
} INK_RECORD_HEADER32, FAR *P_INK_RECORD_HEADER32;




/*************************/
/* REFERENCE SECTION 5.0 */
/*************************/

/*--------------------------------------------------------------------
** A bundle of ink consists of an INK_BUNDLE_RECORD, a series of records,
** terminated with an INK_END_RECORD.
**
** An INK_BUNDLE_RECORD, along with a matching INK_END_RECORD, are the
** mandatory records in the format.  The ink data must start with an
** INK_BUNDLE_RECORD.
**
```

```
** It is suggested that anyone reading this format do a number of validity
** checks on the first record in any ink data. The first record should meet
** the following minimum requirements:
**
** 1)  header.recordType == INK_RECORD_BUNDLE
** 2)  header.recordLength >= inkRecordBundleSize (See general notes in
**      reference section 1.0 for important information about record sizes.)
** 3)  compactionType is an expected and supported value
** 4)  penUnitsPerX and penUnitsPerY seem reasonable and expected:
**      greater than, say, 1000 units per meter (25.4/inch), less than, say,
**      400,000 (~10,000 units per inch)
**
**------------------------------------------------------------*/

typedef struct tag_INK_END_RECORD {
   INK_RECORD_HEADER0 header;        // value is inkRecordEnd
} INK_END_RECORD, FAR *P_INK_END_RECORD;
#define inkRecordEndSize     (inkRecordHeaderLength(inkRecordEnd))



/*************************/
/* REFERENCE SECTION 6.0 */
/*************************/

/*------------------------------------------------------------
** The terms compression and compaction are used somewhat interchangeably
** in this specification but they actually have slightly different meanings
** and are both supported to a certain extent by Jot.
**
** Compression refers to a technique of encoding data such that the resuling
** data, while smaller, is still whole. That is, compression under Jot is
** loss-less. Compaction refers to a process where certain pieces of less
** important data are actually omitted from the stream and are possibly
** reconstructed by the reader of the data..
**
** Using Jot, a writing application may choose to compress only, compact only
```

```
** or use some combination.  The standard compression mechanism defined here
** and implemented in the sample code supports both notions.
**
**-------------------------------------------------------------------*/

typedef U8 INK_COMPACTION_TYPE, FAR *P_INK_COMPACTION_TYPE;
#define INK_COMPACTION_TYPE_SIZE U8_SIZE
#define inkNoCompression          (0)
#define inkStdCompression         (1)

// Other compression schemes may be adopted in future revisions of this
// specification.
```

```
/*************************/
/* REFERENCE SECTION 7.0 */
/*************************/

/*-------------------------------------------------------------------
** The INK_BUNDLE_FLAGS contain some flags that apply to an entire bundle.
** If you wanted to store several pieces of ink that had different
** INK_BUNDLE_FLAGS, you would do it by storing several different bundles.
**
** Advisory flags:
**
** inkPointsRemoved
**     Indicates whether all original points are still present or whether
**     some points were removed to save space.  For applications that are
**     only interested in the visual aspects of ink, many points can be
**     removed that do not affect the appearance (i.e. duplicate points,
**     collinear points, points which deviate less than some screen
**     resolution, etc..).  Some other types of applications must know that
**     points are present at some consistent sampling rate (i.e. some forms
**     of handwriting translation).  This flag indicates whether all
**     original points are still there.
**
**     Note:
**     The purpose of "inkPointsRemoved" is to indicate that the timing
**     information cannot be accurately derived by counting points:
**     replacing individual points with an "elided point" item does not
**     constitute removing points. ("Elided" means omitted or skipped).
**
** inkProxDataRemoved
**     Indicates that the original points between strokes (proximity) were
**     removed to save space.  An out-of-prox point should be stored between
**     strokes to delimit them.  Some applications depend on knowing the
**     time between strokes or at the ends of strokes for certain
**     functions.
**
**     Note:
**     "Proximity" is defined as the stylus being close enough to the tablet
```

** for the tablet to report the stylus position, although perhaps at
** lower accuracy and perhaps at a lower number of points per second. A
** recommended practice is to include "out of proximity" points in the
** recorded ink data when they are used as part of determining the
** amount of time a stylus was out of contact with the tablet, or for
** triggering the completion of an action such as a "gesture".
**
** inkStrokeLimitsPresent
** Indicates that INK_BUTTONS items are also present, and that they
** indicate what the storing app decided the stroke start/end points
** were. (Note: the reading application may otherwise use a different
** algorithm for using tip force values to delimit strokes.)
**
** Note:
** If inkStrokeLimitsPresent is set, then inkButtonDataPresent must also
** be set.
**
** Data flags:
**
** inkAngleDataPresent indicates angle data is present.
** inkForceDataPresent indicates force data is present.
** inkProxDataPresent indicates points are present when pen is lifted
** up (i.e. the force drops below some threshold).
** inkRotationDataPresent indicates pen rotation data is present.
** inkHeightDataPresent indicates pen height data is present.
** inkButtonDataPresent indicates "button state" information is present.
** inkPreMultiplyScale indicates that scaling should be applied before
** the offset value is added ("pre-multiply")
** rather than after ("post-multiply")
**
** Note:
** A previous draft version included a provision for compacting data to an
** approximation based on Bezier curves. Initial results did not show
** promise in terms of efficiency and performance.
**
** "inkBezierRepresentation" would have indicated that the X/Y ordinates
** reflected a Bezier approximation to the original tablet data. This would

```
** have meant that the ordinate data represented aggregates of anchor points
** and control points for each piece wise approximation, and therefore could
** not be used directly to render the data. The definition of these anchor
** and control points, and the example code for the approximation and
** regeneration of the "true" coordinates could not be worked out at this
** time.
**
** Some standard values for pen units per meter follow:
**
** 1000 points per inch digitizer == 39370 pen units per meter
**  500 points per inch digitizer == 19685 pen units per meter
**  200 points per inch digitizer ==  7874 pen units per meter
**  254 points per inch (1/10 mm) == 10000 pen units per meter
**
** 1000 pen units per meter is a reasonable minimum; 400,000 is a reasonable
** maximum value.
**
** The specific format for each of these types of data is described in the
** INK_PENDATA_RECORD documentation (reference section 8.0).
**
** Note:
** The order in which these flags are defined has nothing to do with the
** order in which the data appears in the INK_POINT structure when reading
** or writing point data. For more information, see reference section 21.0.
**
**-----------------------------------------------------------------*/

typedef U16 INK_BUNDLE_FLAGS, FAR *P_INK_BUNDLE_FLAGS;
#define INK_BUNDLE_FLAGS_SIZE U16_SIZE
#define inkPointsRemoved          (flag0)
#define inkProxDataRemoved        (flag1)
#define inkAngleDataPresent       (flag2)
#define inkForceDataPresent       (flag3)
#define inkRotationDataPresent    (flag4)
#define inkHeightDataPresent      (flag5)
#define inkButtonDataPresent      (flag6)
#define inkStrokeLimitsPresent    (flag7)
```

```
#define inkPreMultiplyScale        (flag8)

// Reserved: flag9, flag10, flag11, flag12, flag13, flag14, flag15.
// More flags beyond flag15 can be added in a new record type
// in a later revision to this specification.

typedef struct tag_INK_BUNDLE_RECORD {
   INK_RECORD_HEADER8     header;        // value is inkRecordBundle
   U8                     version;       // Value for release 1.0 is 1
   INK_COMPACTION_TYPE    compactionType;
   INK_BUNDLE_FLAGS       flags;         // flags for the whole bundle
   U32                    penUnitsPerX;  // pen units per meter (x dir)
   U32                    penUnitsPerY;  // pen units per meter (y dir)
} INK_BUNDLE_RECORD, FAR *P_INK_BUNDLE_RECORD;

#define inkPointDefaultVersion          (1)
#define inkPointDefaultCompactionType   (inkStdCompression)
#define inkPointDefaultBundleFlags      (0)
#define inkPointDefaultPenUnitsPerX     (1000)
#define inkPointDefaultPenUnitsPerY     (1000)

#define inkRecordBundleSize \
   (inkRecordHeaderLength(inkRecordBundle) +  U8_SIZE + \
    INK_COMPACTION_TYPE_SIZE + INK_BUNDLE_FLAGS_SIZE +    \
    U32_SIZE + U32_SIZE)


/*************************/
/* REFERENCE SECTION 8.0 */
/*************************/

/*---------------------------------------------------------------
** A penData record contains the actual pen data for one or more pen strokes.
** The bounds applies to all the strokes contained within this record.
** Multiple strokes are typically grouped into one record to increase the
** efficiency of the compression algorithm, though strokes may be stored
```

```
** individually, if desired.
**
** The bounds is the pure mathematical bounds of the raw pen points and does
** not take into account any rendering information such as the pen tip or the
** line width.  All points in the INK_PENDATA have been normalized relative
** to the lower bounds in the INK_PENDATA header.
**
** Some applications will prefer to know the bounds of individual strokes.
** This can be accomplished in two ways.
**
** 1) The bounds for a given stroke can be computed when reading the file
**    by decompressing an INK_PENDATA_RECORD into its strokes and then
**    traversing the points in each stroke to build the bounds for each
**    stroke.
**
** 2) An application can decide to store only one stroke per
**    INK_PENDATA_RECORD (and thus the bounds of the PENDATA_RECORD is
**    already the bounds of one stroke).  The sacrifice here is in
**    compression efficiency and the need to still support reading files
**    written by other applications that might group multiple strokes
**    into a single INK_PENDATA_RECORD.
**
** Note:
** In practice, our experience is that unpacking the data in order to compute
** the bounds for each stroke to check for strokes that intrude into a given
** region is not an excessive burden.  The checks that would have been done
** on the bounds of each stroke can be done on the builds for each penData
** group, and not all strokes must be checked individually.
**
** The format of the pen data is determined by the settings for
** compactionType and flags in the INK_BUNDLE_RECORD structure, and
** is described later in this file.  Two formats are currently defined:
** an uncompacted format and a delta-encoded compacted format, both with
** optional components present or absent depending on the state of the flags
** in the INK_BUNDLE_RECORD.
**
**------------------------------------------------------------------*/
```

```
typedef struct tag_INK_PENDATA_RECORD {
   INK_RECORD_HEADER32 header;    // value is inkRecordPenData
   RECT32           bounds;
   U8               inkData[1]; // ink data goes here: definitions
                        // follow later in this file.
} INK_PENDATA_RECORD, FAR *P_INK_PENDATA_RECORD;
#define inkRecordPenDataSize(data_length)  \
  (inkRecordHeaderLength(inkRecordPenData) +  RECT32_SIZE + (data_length))


/*************************/
/* REFERENCE SECTION 9.0 */
/*************************/

/*-----------------------------------------------------------------
** Ink scale is recorded in two fixed point values.  A unity scale (scale
** of one) is represented as 0x00010000, a scale of 0.5 as 0x00008000.
**
** Note:
** All ink is located relative to the lower-left (0,0) corner of a logical
** page or window.  Scale and offset operations are cumulative, much in the
** same way as in PostScript.  One begins with a normalized graphics state
** and sequentially applies the scale and offset operations to that matrix.
** The INK_SCALE_RESET record returns the graphics state to its default state
** (i.e., the transformation matrix is set to an identity matrix and the
** offset is reset to the default of 0).  By default, scaling is applied
** after adding in any offset specified in an INK_OFFSET_RECORD.  If the ink
** bundle has the inkPreMultiplyScale bit set, for all ink in that bundle
** scaling is applied before adding in any offset.
**
** As used in this format, ink scale and offset values are set by the storing
** application, to be applied by the rendering application.  If the storing
** application collected the ink at scales of (2.0,2.0), the storing
** application should insert an INK_SCALE_RECORD with a scale of (0.5,0.5)
** for the rendering application to multiply all ink X and Y coordinates by.
```

```
**
** It is the responsibility of the storing application to deal with any
** effects from round-off or truncation error due to the limits of precision
** in the FIXED_FRACTION values used in INK_SCALE_RECORDs.
**
** An ink scale record indicates a scale change that stays in effect until
** another ink scale record is encountered. Ink scale values compound: if
** the current scale is (2.0,2.0) and an INK_SCALE_RECORD is encountered with
** scale of (2.0,3.0), the scale to be applied to ink then becomes(4.0,6.0).
** In absence of any ink scale record, the default ink scale is unity. In
** general, a typical usage pattern for an application that supports drawing
** ink while zoomed at scale is to record a number of strokes at a given
** scale, reset the scale with an INK_SCALE_RESET_RECORD (which resets both
** the scale and the offset to the default values), then switch to another
** scale, then record a number more strokes, and so on.
**
** Note:
** The extension scaling and offset to the Z ordinate value is not defined in
** this version of the specification. The extension to Z scaling and offset
** in a "standard" record type (i.e. not an application-specific record) may
** be addressed in the future.
**
**---------------------------------------------------------------------*/

typedef struct tag_INK_SCALE {
   FIXED_FRACTION     x;    // scale in the x direction
   FIXED_FRACTION     y;    // scale in the y direction
} INK_SCALE, FAR *P_INK_SCALE;
#define INK_SCALE_SIZE (FIXED_FRACTION_SIZE+FIXED_FRACTION_SIZE)

#define inkPointDefaultScale    (INK_UNITY_SCALE)     // Unity.

typedef struct tag_INK_SCALE_RECORD {
   INK_RECORD_HEADER8  header;     // value is inkRecordScale
   INK_SCALE           scale;
} INK_SCALE_RECORD, FAR *P_INK_SCALE_RECORD;
#define inkRecordScaleSize \
```

```
(inkRecordHeaderLength(inkRecordScale) + \
 FIXED_FRACTION_SIZE + FIXED_FRACTION_SIZE)



/**************************/
/* REFERENCE SECTION 10.0 */
/**************************/

/*----------------------------------------------------------------
** The offset position is used to relocate ink data, after scaling.  For
** example, in a forms application, ink in a sketch field is drawn relative
** to a given sketch field in the form.  The location of this original field
** is important to know so we know how the ink in this bundle relates to its
** original field.  If we wanted to move this ink to another field (i.e.
** cut/paste or move), we would need to know the location of the original
** field so we could render the ink in the new field in a manner consistent
** with how it was drawn relative to its original field (i.e. a similar
** baseline for a hand-written signature).
**
** This record is optional.  If it exists, it will then apply to all
** following pen data in the file.  If it is not present it is assumed that
** no information of this type is relevant.  For example, while field ink
** would have an offset position, markup ink over an entire form would not
** have a offset position (or would have an offset position of (0,0) and a
** scale of (1,1)) because it is relative to the entire form coordinate
** system, not relative to some piece in the form.
**
** Note:
** This approach allows a reader to "blindly" apply the scale and offset
** values specified to ink data, and puts the burden for computing
** compounding of multiple zoom levels, etc., on the writing application.
**
**----------------------------------------------------------------*/

typedef struct tag_INK_OFFSET_RECORD {
```

```
    INK_RECORD_HEADER8 header;        // value is inkRecordOffset
    XY32            positionOffset; // values are in pen units
} INK_OFFSET_RECORD, FAR *P_INK_OFFSET_RECORD;

#define inkRecordOffsetSize \
    (inkRecordHeaderLength(inkRecordOffset) + XY32_SIZE)

#define inkPointDefaultOffset   ((S32) 0)   // No offset.


typedef struct tag_INK_SCALE_RESET_RECORD {
    INK_RECORD_HEADER0 header;        // value is inkRecordScaleReset
} INK_SCALE_RESET_RECORD, FAR *P_INK_SCALE_RESET_RECORD;

#define inkRecordScaleResetSize  (inkRecordHeaderLength(inkRecordScaleReset))




/**************************/
/* REFERENCE SECTION 11.0 */
/**************************/

/*-------------------------------------------------------------------
** Ink color is represented as an rgb value, plus opacity.
**
** The default color is black (r,g,b,o) = (0,0,0,255).  A color change
** present in the file remains in effect until another color change.
** Typically, the color will stay the same for many ink strokes and thus
** a color record will only be used occasionally when the color changes.
**
** "Opacity" is rather vaguely understood, especially in color environments.
** In this context, opacity means the degree to which the display underneath
** the ink shows through.  An opacity value of 255 means that nothing under
** the ink shows through; 0 means that everything shows through (the ink
** is transparent).  It is up to the reading application to define the
** implementation of opacity on the reading platform.
```

```
**
** The color/opacity value of (255,255,255,0), or "transparent white" is
** defined as an "erase" color. In inking applications that support a true
** "erase" function, such as the ability to erase annotation ink on an
** "original" document (perhaps a FAX image) the "erase" color restores the
** background image where painted. The "background image" is defined by the
** rendering application.
**
** Applications which do not support a true "erase" function may interpret
** this as some other drawing function, such as drawing the "background"
** color.
**
**---------------------------------------------------------------*/

typedef union {
 U32    all;
 struct {
  U8   red,
       green,
       blue,
       opacity;   // opaqueness: see defines below
 } rgb;
} INK_COLOR, FAR *P_INK_COLOR;
#define INK_COLOR_SIZE (U32_SIZE)

typedef struct tag_INK_COLOR_RECORD {
   INK_RECORD_HEADER8  header;        // value is inkRecordColor
   INK_COLOR           color;
} INK_COLOR_RECORD, FAR *P_INK_COLOR_RECORD;
#define inkRecordColorSize \
     (inkRecordHeaderLength(inkRecordColor) + U32_SIZE)

// Standardized opacity values:
// A recommended practice is that an opacity value of 128 (midway between
// 0 and 255) be used for "highlighter" colors. A recommended practice is
// that grey values as defined below be used for "standard grey"
// highlighters.
```

```
#define inkOpacityTransparent  0x00
#define inkOpacityHighlight    0x80
#define inkOpacityOpaque       0xFF

// Standard solid colors:

#define inkColorErase      {0xFF,0xFF,0xFF,0x00}
#define inkColorWhite      {0xFF,0xFF,0xFF.0xFF}
#define inkColorLtGrey     {0x80,0x80,0x80,0xFF}
#define inkColorDkGrey     {0x40,0x40,0x40,0xFF}
#define inkColorBlack      {0x00,0x00,0x00,0xFF}

// Standard highlighter (transparent) colors:

#define inkColorLtGreyHighlight {0x80,0x80,0x80,0x80}
#define inkColorDkGreyHighlight {0x40,0x40,0x40,0x80}

#define inkDefaultColor ((INK_COLOR) inkColorBlack)



/***************************/
/* REFERENCE SECTION 12.0 */
/***************************/

/*-----------------------------------------------------------------
** Time is measured in milliseconds.
**
** Note:
** Because of the difficulty synchronizing clocks on different machines
** at the time granularity of digitizing tablets, and because the "editing"
** of ink at a later time makes the definition of the absolute time for each
** ink point ambiguous, the base for the time is arbitrary. All times in
** strokes are just relative to each other with no absolute time
** relationship.
```

```
**
** These records, when encountered in the file, apply to the next stroke data
** in the file (so this record comes before the penData that it applies to).
** End time records are not required. The interpretation of an end time
** which is in conflict with the end time inferred from the assumed data rate
** of points and the number of points (including elided points) is not
** defined.
**
** Start time is the time for the first point in the following penData record
** and end time is the time of the last point in the following penData
** record, because if you are recording tip force, the exact definition of
** pen up and pen down may be fuzzy and/or application dependent.
**
**--------------------------------------------------------------*/

typedef U32 INK_TIME, FAR *P_INK_TIME;     // milliseconds
#define INK_TIME_SIZE U32_SIZE

#define inkDefaultTime     ((INK_TIME) 0)

typedef struct tag_INK_START_TIME_RECORD {
   INK_RECORD_HEADER8 header;       // value is inkRecordStartTime
   INK_TIME         startTime;
} INK_START_TIME_RECORD, FAR *P_INK_START_TIME_RECORD;
#define inkRecordStartTimeSize \
     (inkRecordHeaderLength(inkRecordStartTime) + INK_TIME_SIZE)

typedef struct tag_INK_END_TIME_RECORD {
   INK_RECORD_HEADER8 header;       // value is inkRecordEndTime
   INK_TIME         endTime;
} INK_END_TIME_RECORD, FAR *P_INK_END_TIME_RECORD;
#define inkRecordEndTimeSize   \
     (inkRecordHeaderLength(inkRecordEndTime) + INK_TIME_SIZE)
```

```
/*************************/
/* REFERENCE SECTION 13.0 */
/*************************/

/*---------------------------------------------------------------
** INK_PENDATA_RECORDs can be grouped.  If they are grouped, each
** INK_PENDATA_RECORD can be assigned a group number.  All
** INK_PENDATA_RECORDs with the same group number belong to the same group.
**
** The exact interpretation of grouping is up the applications involved.
** Writing applications may group ink data, but not all reading applications
** that read the data may interpret grouping in the same way.
**
** For example, grouping could be used in the traditional fashion as in
** drawing programs so the user moves or copies an entire group of
** INK_PENDATA_RECORDs together.  A group could also be used to signify a
** series of INK_PENDATA_RECORDs entered by the user all within some criteria
** (i.e. all during one proximity session or all within some time frame).
**
** Group numbers are simply signed 16 bit values and can be anything.  They
** do not need to be contiguous (i.e. they do not need to be 0,1,2).  They
** can be 12,49,-12345 if that is useful.
**
** This record can also be used as a simple marker for starting a new group
** when the groupId is not really used: Group numbers of 0,0,0,0 ... are
** thus permitted.
**
** INK_GROUPs are nestable.  Group 0 is reserved as the end-of-group marker
** for disjoint groups.  If no end-of-group marker is encountered before the
** end of the file or the end of all ink data (as indicated by an
** INK_END_RECORD), all current (and possibly nested) groups are terminated
** as if end-of-groups markers for them had been encountered.
**
**---------------------------------------------------------------*/

typedef S32 INK_GROUP, FAR *P_INK_GROUP;
#define INK_GROUP_SIZE S32_SIZE
```

```
typedef struct tag_INK_GROUP_RECORD {
   INK_RECORD_HEADER8  header;       // value is inkRecordGroup
   INK_GROUP          groupId;   // application-specific interpretation
} INK_GROUP_RECORD, FAR *P_INK_GROUP_RECORD;

#define inkDefaultGroup ((INK_GROUP) 0)

#define inkRecordGroupSize  \
     (inkRecordHeaderLength(inkRecordGroup) + INK_GROUP_SIZE)




/**************************/
/* REFERENCE SECTION 14.0 */
/**************************/

/*----------------------------------------------------------
** Some applications may support the idea of rendering ink as if it were
** drawn by a certain shaped pen tip.  The most common pen tips would be
** round or rectangular.  The exact details of how to render a given pen
** tip will be application specific, but this record states what pen tip
** parameters were used by the storing app.
**
** Pen tips determine, in part, how ink is rendered.  For pen tip types
** defined in future versions of this format which require additional
** parameters (such as the X and Y rectangle for a simulated nib pen, or
** brush dimensions for a simulated brush), additional data is included
** at the end of the structure.
**
** The writing application should be aware that the reading application will
** only do "the best possible" job of rendering and that fully compliant
** reading applications may not be able to render certain nib types and/or
** colors.  Both reading and writing applications should pay particular
** attention to the following notes regarding defaults and ink drawn at a
** width of zero.
```

```
**
** A pen tip which is drawing ink in zero width renders at the minimum
** visible width the reading application will support.
**
** A recommended practice is that ink which should not render (should this
** be called for) be drawn with a color value of (0,0,0, 0), i.e., black,
** completely transparent.
**
** Pen tip size should scale when an INK_SCALE_RECORD is encountered. The
** writing application should write a new INK_PENTIP_RECORD after an
** INK_SCALE_RECORD if the writing application does not want the pen tip
** size to scale along with the ink. If the pen tip scales to zero width,
** it should be rendered by the reading application according to the comment
** above.
**
** The default pen tip if no pentip record exists is INK_PENTIP_ROUND, with a
** width of one twip. The dimensions of a round nib specify diameter, not
** radius: the X/Y coordinate is the center of this diameter. Similarly, for
** for rectangular nibs, the X/Y coordinate is the center of the rectangle.
**
** Note:
** This specification does not specify information for an algorithmic
** variation in nib width, ink color, or other "brush" effects as a function
** of tip force, speed or any other factor. An example would be for an
** application to draw wider ink as the user presses down harder with the
** stylus. Applications wishing to implement such features may do so using
** application-specific record types for this revision of the specification.
**
**-----------------------------------------------------------------------*/

typedef S16 INK_PENTIP, FAR *P_INK_PENTIP;
#define INK_PENTIP_SIZE S16_SIZE
#define INK_PENTIP_ROUND          (0) // Diameter in twips
#define INK_PENTIP_RECTANGLE      (1) // Dimensions in twips
#define INK_PENTIP_SLANT_RECTANGLE (2)
#define INK_PENTIP_ROUND_FLAT_END (3)
// ... more to be filled in here if needed
```

```
#define inkDefaultPentip        INK_PENTIP_ROUND
#define inkDefaultPentipData    ((U16) 1)

typedef struct tag_INK_PENTIP_SLANT {
  SIZE16  rectangle_size; // INK_PENTIP_SLANTRECTANGLE
  U16     angle;          // Whole degrees from vertical, counter-clockwise
} INK_PENTIP_SLANT, FAR *P_INK_PENTIP_SLANT;

typedef union {
  U16              round_width;      // INK_PENTIP_ROUND
  SIZE16           rectangle_size;   // INK_PENTIP_RECTANGLE
  INK_PENTIP_SLANT  slant;           // INK_PENTIP_SLANT_RECTANGLE
  U16              round_flat_width; // INK_PENTIP_ROUND_FLAT_END
} INK_PENTIP_DATA, FAR *P_INK_PENTIP_DATA;

// Size of the union is determined by INK_PENTIP_SLANT
#define INK_PENTIP_DATA_SIZE (SIZE16_SIZE+U16_SIZE)

typedef struct tag_INK_PENTIP_RECORD {
  INK_RECORD_HEADER8  header;      // value is inkRecordTip
  INK_PENTIP          tip;
  INK_PENTIP_DATA     tip_data;
} INK_PENTIP_RECORD, FAR *P_INK_PENTIP_RECORD;
#define inkRecordTipSize   \
    (inkRecordHeaderLength(inkRecordTip) + INK_PENTIP_SIZE + \
     SIZE16_SIZE + U16_SIZE)



/***************************/
/* REFERENCE SECTION 15.0 */
/***************************/

/*------------------------------------------------------------
** For some applications, it will be important to know the sampling rate of
```

```
** the pen digitizer.
**
** This record would likely be present once in a bundle and would typically
** be after the INK_BUNDLE_RECORD, but before the first pen data.
**
** Note:
** Writing applications are not required to report the "true" sampling rate
** of the digitizer, nor are rendering applications required to play back the
** ink at the specified rate. It is likely that most types of rendering
** applications will render ink as rapidly as possible to construct a display
** in minimum time, and that some types of animation applications will
** intentionally set an arbitrary sampling rate to vary the display rate.
**
** Note:
** For hardware which supports a highly variable sampling rate, the writing
** application can simulate a very high sampling rate (say, 1000 points/
** second), and use skip records for "elided" points to achieve an exact time
** value (at 1-millisecond resolution) for each point.
**
** A default value for sampling rate has been arbitrarily defined below.
**
**-----------------------------------------------------------------------*/

typedef struct tag_INK_POINTS_PER_SECOND_RECORD {
   INK_RECORD_HEADER8  header;        // value is inkRecordPointsPerSecond
   U16                 pointsPerSecond;
} INK_POINTS_PER_SECOND_RECORD, FAR *P_INK_POINTS_PER_SECOND_RECORD;

#define inkPointDefaultPointsPerSecond  (100)

#define inkRecordPointsPerSecondSize   \
     (inkRecordHeaderLength(inkRecordPointsPerSecond) + U16_SIZE)



/**************************/
```

```
/* REFERENCE SECTION 16.0 */
/**************************/

/*--------------------------------------------------------------------
** Units for Z height of stylus above the tablet.
**
** This record would only be present once in a bundle and would typically be
** after the INK_BUNDLE_RECORD, but before the first pen data.
**
**-------------------------------------------------------------------*/

typedef struct tag_INK_UNITS_PER_Z_RECORD {
   INK_RECORD_HEADER8  header;      // value is inkRecordUnitsPerZ
   U32                 unitsPerZ;   // pen units per meter (Z height)
} INK_UNITS_PER_Z_RECORD, FAR *P_INK_UNITS_PER_Z_RECORD;

#define inkPointDefaultUnitsPerZ  (10000)  // 0.1 mm units

#define inkRecordUnitsPerZSize \
     (inkRecordHeaderLength(inkRecordUnitsPerZ) + U32_SIZE)



/**************************/
/* REFERENCE SECTION 17.0 */
/**************************/

/*--------------------------------------------------------------------
** Units for stylus tip force.
**
** This record would only be present once in a bundle and would typically be
** after the INK_BUNDLE_RECORD, but before the first pen data.
**
** Note:
** This specification assumes some level of accuracy and linearity for tip
** force data, if such data is present. However, since tip force sensors in
```

```
** current digitizer tablet and stylus designs may well vary in accuracy and
** linearity from one unit to the next even for hardware of the same design
** and model, and since algorithms for using tip force to determine stroke
** start and end are likely to differ, a recommended practice for writing
** applications that use the tip force value to determine the "touch" points
** in a stroke is to mark those points using the touch bit in the INK_BUTTONS
** structure.
**
** It is also recommended that vendors supporting tip force sensing in their
** hardware linearize their transducers to the greatest extent possible.
**
** Because of the likelihood that tip force transducers may not be accurately
** linearized, negative tip force values, while perhaps somewhat absurd
** are possible and are permitted in this specification.
**
**---------------------------------------------------------------*/

typedef struct tag_INK_UNITS_PER_FORCE_RECORD {
   INK_RECORD_HEADER8  header;        // value is inkRecordUnitsPerForce
   U32                 unitsPerForce; // tip-force units per k-gram of force
} INK_UNITS_PER_FORCE_RECORD, FAR *P_INK_UNITS_PER_FORCE_RECORD;

#define inkPointDefaultUnitsPerForce  (1000)    // grams of force

#define inkRecordUnitsPerForceSize \
     (inkRecordHeaderLength(inkRecordUnitsPerForce) + U32_SIZE)



/*************************/
/* REFERENCE SECTION 18.0 */
/*************************/

/*---------------------------------------------------------------
** The INK_APP_RECORD record is a universal record to be used by individual
** applications to put data into the file that is not supported by an
```

```
** additional publicly defined record type. The basic idea is that an
** application puts its own unique application signature into the appData
** bytes in the INK_APP_RECORD. This identifies the data as originating with
** a particular application. Then, an application defines a set of
** subRecordTypes that they wish to use. Then, using these subRecordTypes
** they can put a wide variety of information into the file. By examining
** the appData signature and comparing it to theirs, an application can
** decide whether it knows how to interpret the various subRecordtypes.
**
**-------------------------------------------------------------------*/

typedef struct tag_INK_APP_RECORD {
   INK_RECORD_HEADER32 header;        // value is inkRecordApp
   U8              appSignature[8];// reserved for possible unique
                            // application signature
   U16             subRecordType;
   // data here appropriate to the subRecordType and appData signature
} INK_APP_RECORD, FAR *P_INK_APP_RECORD;
#define inkRecordAppSize(data_length)   \
   (inkRecordHeaderLength(inkRecordApp) + 8 + U16_SIZE + (data_length))



/***************************/
/* REFERENCE SECTION 19.0 */
/***************************/

/*-------------------------------------------------------------------
** Definition of the inkData components of an INK_PENDATA_RECORD:
**
** Uncompacted point format:
** -----------------------
**
** This structure immediately follows the rest of the INK_PENDATA_RECORD.
** The structure has several optional components, present or not present as
** indicated by the INK_BUNDLE_FLAGS in the INK_BUNDLE_RECORD. The format is
```

```
** a sequence of "point values", each containing all the scalar data for each
** sampled tablet point.
**
** In the uncompacted format, there is a single structure that contains all
** of the state information for each point from the tablet.  Components not
** present (as indicated by the INK_BUNDLE_FLAGS) are just that: not present,
** do not exist, do not occupy space.
**
** Compacted point format:
** ---------------------
**
** In the compacted format, "State values", such as the stylus state of
** touch/no-touch/out-of-prox or the on/off state of the barrel switches, are
** stored in a compacted "INK_BUTTONS" item (represented using reserved
** encodings in the INK_POINT coordinate values) interjected when their state
** changes.  The initial state is assumed to be "not touching", "out of
** proximity", all barrel switches "off".  It is possible to have both tip
** force data, and explicit starts and ends of strokes: the starts and ends
** of the strokes are then points that were considered to be such by the
** original application storing the data.    The INK_BUTTONS record reflects
** the state of the next X/Y point following.
**
**------------------------------------------------------------------*/
```

```
/**************************/
/* REFERENCE SECTION 20.0 */
/**************************/

/*--------------------------------------------------------------
** "INK_BUTTONS" items may most often be used only to indicate stylus touch
** and out-of-prox state, and perhaps a single barrel button. The format is
** optimized for this case. The format extends to a total of 28 stylus/puck
** buttons.
**
**-------------------------------------------------------------*/

// The lowest-order bit (flag0) is "0" when the stylus is out of
//   proximity, "1" when it is in proximity.
// Second lowest-order bit (flag1) is "1" to indicate the next inkPoints are
//   when the stylus is touching (the start of a stroke: tip-switch "on"),
//   "0" to indicate that the stylus is not touching (end of a stroke).
// Third bit (flag2) indicates state of first (or only) barrel switch,
//   etc.
// 31'st bit (flag30) is normally "0", "1" is indicates there are more
//   than 29 barrel/puck buttons with state, and the rest are in the next
//   four-byte word.

typedef U32 INK_BUTTONS, FAR * P_INK_BUTTONS;

// These definitions hold the maximum and minimum values that
// can be used with the S15 and S31 representations described in
// this document:

#define MAX_S31 ((S32) 0x3FFFFFFF)
#define MIN_S31 ((S32) 0xC0000000)

#define MAX_S15 ((S16) 0x3FFF)
#define MIN_S15 ((S16) 0xC000)

#define MAX_S7 ((S16) 0x003F)
#define MIN_S7 ((S16) 0xFFC0)
```

```
#define MAX_S3 ((S16) 0x0003)
#define MIN_S3 ((S16) 0xFFFC)

// SignExtend4/8/16/32: Sign-extend an S3, S7, S15, S31 to an S32:

#define SignExtend4(value)  ((S32)                    \
  (((value)&0x00000004l)== 0 ?  ((value)&0x00000007l)    \
                    : (((value)&0x00000007l) | 0xFFFFFFF8l)))

#define SignExtend8(value)  ((S32)                    \
  (((value)&0x00000040l)== 0 ?  ((value)&0x0000007Fl)    \
                    : (((value)&0x0000007Fl) | 0xFFFFFF80l)))

#define SignExtend16(value) ((S32)                    \
  (((value)&0x00004000l)== 0 ?  ((value)&0x00007FFFl)    \
                    : (((value)&0x00007FFFl) | 0xFFFF8000l)))

#define SignExtend32(value) ((S32)                    \
  (((value)&0x40000000l)== 0 ?  ((value)&0x7FFFFFFFl) \
                    : (((value)&0x7FFFFFFFl) | 0x80000000l)))


/***************************/
/* REFERENCE SECTION 21.0 */
/***************************/

/*---------------------------------------------------------------------
** INK_POINT data.  The INK_POINT structure varies in size depending on
** flags set in the bundle header.  The XY32 position is always present, but
** the force, rho, height, angle, and buttons members are only present when
** indicated by the corresponding flag in the bundle header.  When optional
** data is present, it is present in the order defined by this structure;
** that is, position, force, height, rho, angle, and finally buttons.
**
** The INK_POINT structure has the following elements:
```

```
**
** position - required and always present
**   Positions are measured with (0,0) at the lower-left, X increasing to
**   the right, Y increasing upwards.  Values are actually S31, not S32.
**   The high bit in X and Y must be zero.
** force    - optional, present if inkForceDataPresent is asserted
**   Units are in pen force units, zero is no contact.
** height   - optional, present if inkHeightDataPresent is asserted
**   Units are in pen unitsPerZ as specified by inkPointDefaultUnitsPerZ or
**   by an INK_UNITS_PER_Z_RECORD, whichever is appropriate.  Units increase
**   as the stylus is taken away from the tablet.  Zero means "just in
**   contact".  Negative values could possibly result from spring action if
**   the stylus is pressed hard, or if the tablet is not perfectly accurate.
** rho      - optional, present if inkRotationDataPresent is asserted
**   Angles are measured in degrees from some nominal orientation of
**   "stylus button on top" (somewhat arbitrary).  Angles increase with
**   clockwise rotation as seen from the rear end of the stylus.
** angle    - optional, present if inkAngleDataPresent is asserted
**   Angles are measured in pen angle units from the vertical.  Theta
**   increases in the positive-X direction, phi in the positive-Y.
** buttons  - optional, present if inkButtonDataPresent is asserted
**
```

```
** When the INK_BUNDLE_RECORD member compactionType is inkStdCompression,
** all data in this structure is compressed according to the methods
** described in reference section 23.0. For more details on how to interpret
** the compressed data stream, see the sample code. The bundle flags which
** indicate whether a particular piece of data is present are used regardless
** of whether the data is compressed or not. Note that when data is written
** in compressed format, it is NOT written in Intel order but rather most
** significant byte first. In compressed form, some of the eight bit delta
** values are reserved for button data and elided (skipped) point counts.
** This has two important ramifications. 1) When expecting a point,
** compacted button data or elided point data may be encountered instead, and
** 2) when the inkButtonDataPresent flag is asserted in the bundle header,
** button data will appear in the place of a point and not in addition to a
** point. If inkButtonDataPresent is not asserted, the reader need not check
** the point data for the special case of button data; however, the point
** data must still be checked to see if it is a count of elided points rather
** than an actual point.
**
**----------------------------------------------------------*/

typedef struct tag_INK_POINT {
   XY32      position;  // required x/y point data
   S16       force;     // optional force data
   S16       height;    // optional z height data
   S16       rho;       // optional rotational data
   ANGLE16   angle;     // optional theta and phi data
   INK_BUTTONS buttons; // optional proximity, contact, button data
} INK_POINT, FAR *P_INK_POINT;
```

```
/*************************/
/* REFERENCE SECTION 22.0 */
/*************************/

/*-----------------------------------------------------------------
** The following default values are assumed before the start of any
** INK_BUNDLE:
**
**-----------------------------------------------------------------*/

#define inkPointDefaultXYPosition   ((S32) 0)
#define inkPointDefaultForce        ((S16) 0)
#define inkPointDefaultHeight       ((S16) 0)
#define inkPointDefaultRho          ((S16) 0)
#define inkPointDefaultAngle        ((S16) 0)
#define inkPointDefaultButtons      ((U32) 0)


/*************************/
/* REFERENCE SECTION 23.0 */
/*************************/

/*-----------------------------------------------------------------
** Compacted point format:
** --------------------
**
** A recommended practice is always to use the compacted point format, not
** the uncompacted point format.  Sample code for reading and writing the
** compacted format is included in an appendix.
**
** This structure also immediately follows the rest of the
** INK_PENDATA_RECORD.
**
```

** The uncompacted values above are stored in sequential bytes in a more
** compact, delta-oriented format. Deltas are all signed values, a value to
** add to the previous value. The first point in an INK_PENDATA_RECORD is
** always relative to the defined default values for each component of the
** point.
**
** The storing application, as an alternative to eliminating points, can
** specify a "skip" record for elided points. The skipRecord indicates that
** a number of points were skipped, and the reading application is free to
** insert values for the elided points (interpolating where appropriate).
** The intent is to allow for accurate time information to be maintained
** between time stamps for synchronization with recorded voice, press-hold
** gesture recognition, etc.
**
** Compacted data is written most significant byte first so that reading
** applications can read the first byte and determine (from the top two bits)
** how large the encoded delta is.
**
** Note:
** "Reserved encodings" are those encodings that, if real points, would fit
** into the next smaller delta size. 16 bit deltas and 8 bit deltas have
** reserved encodings. The reserved encodings for 16 bit deltas are all 16
** bit delta pairs where both X and Y are within the inclusive range MIN_S7
** and MAX_S7. Similarly, the reserved encoding for 8 bit deltas are all 8
** bit delta pairs where both X and Y are within the inclusive range MIN_S3
** and MAX_S3. In revision 1.0 of Jot, three of the reserved encodings for 8
** bit deltas are used for special cases: skip counts (reference section
** 27.0) and button changes (reference section 26.0).
**
** x/y position:
** ------------
**
** 32-bit absolute X/Y:
**
**   Two 32 bit long words: Data is actually two S31s:
**
** |0|0| (30 low-order bits of X) |

```
**        ... Sign bit is taken from first bit of next word.
**    ---------------------------------------------------------
**    |X|    (sign bit of X plus 31 bits of Y)              |
**    ---------------------------------------------------------
**
** 16-bit short delta X/Y:
**
**    Short words: two 16 bit words: Deltas are actually two S15s:
**    Values that would fit into an 8-bit byte delta are reserved.
**
**    |0|1|  (14 low-order bits of delta-X)      |
**        ... Sign bit is taken from first bit of next word.
**    ---------------------------------------------
**    |X|    (sign bit of X plus 15 bits of delta Y |
**    ---------------------------------------------
**
** 8-bit byte delta X/Y:
**
**    Bytes: two bytes: Deltas are actually two S7s:
**    Values that would fit into a 4-bit nibble delta are reserved.
**
**    |1|0|  (6 low-order bits of delta-X)  |
**        ... Sign bit is taken from first bit of next word.
**    --------------------------------------
**    |X|    (7 bits of delta-Y)            |
**    --------------------------------------
**
** 4-bit nibble delta X/Y:
**
**    Nibbles: one byte: Deltas are actually S3:
**
**    |1|1| (S3 delta-X) | (S3 delta-Y) |
**    --------------------------------
**
**-----------------------------------------------------------------*/
```

```
/***************************/
/* REFERENCE SECTION 24.0 */
/***************************/

/*-----------------------------------------------------------------
** Tip force:
** ----------
**
** 16-bit absolute force:
**
**    Short word: one word: Value is actually S15:
**    Values that would fit into an 8-bit byte delta are reserved.
**
**    |0|    (15 bits of force)          |
**    ----------------------------------------
**
** 8-bit byte delta force:
**
**    Byte: one byte: Deltas are actually S7:
**
**    |1| (S7 delta-force)   |
**    -------------------------
**
**
** Height:
** ------
**
** (Same encoding as tip force)
**
** Rho:
** ---
**
** (Same encoding as tip force)
**
**
** Stylus theta-phi:
** ----------------
```

```
**
** 16-bit absolute theta-phi:
**
**    Short words: two words: Data is actually S15:
**
**    |0|0|    (14 low-order bits of theta)    |
**    ------------------------------------------
**           ... Sign bit is taken from first bit of next word.
**    |X|     (15 bits of phi)             |
**    ------------------------------------------
**
**----------------------------------------------------------------------*/
```

```
/**************************/
/* REFERENCE SECTION 25.0 */
/**************************/

/*-----------------------------------------------------------------
** 8-bit byte delta theta-phi
**
**    Bytes: two bytes: Deltas are actually S7:
**    Values that would fit into a 4-bit nibble delta are reserved.
**
**    |0|1|   (6 low-order bits of delta-theta)|
**    ---------------------------------------
**          ... Sign bit is taken from first bit of next word.
**    |X|     (7 bits of delta-phi)          |
**    ---------------------------------------
**
** 4-bit nibble delta theta-phi
**
**    Nibbles: one byte: Deltas are actually S3:
**
**    |1|0|(S3 delta-theta)|(S3 delta-phi)|
**    ---------------------------------
**
** Note:
** Leading bit values of |1|1| are reserved
**
**---------------------------------------------------------------*/
```

```
/**************************/
/* REFERENCE SECTION 26.0 */
/**************************/

/*---------------------------------------------------------------
** Since the X/Y data is always present, we use some of the reserved delta
** encodings to encode button states and elided (skipped) points. We use the
** 8-bit delta encodings that are unused: the values that can fit into the
** smaller 4-bit delta encodings.
**
** Button/tip records:
** ----------------
**
** It is assumed that the state of barrel buttons and the touching sensor on
** the stylus change infrequently. A compacted button/tip record is only
** included when the state changes in one of the switches. The button state
** value applies to the X/Y point immediately following the button state
** record.
**
**
** (Taken from 8-bit byte delta X/Y: two bytes total)
**
**     |1|0| 0|0|0|0|0|0/1|   0|X|.|.|.|.|X|X|
**     --------------------------------------
**           (delta-X)          (delta-Y)
**
** An eight-bit delta with delta-X == 0 or 1, and delta-Y in the range
** (-4..3) indicates a button state encoding.
**
** It is likely to be the case that many hardware platforms have only one
** barrel button.
**
** The three delta-Y bits indicate the "touch", "out-of-prox", and "first
** barrel button" state as follows:
**
** low-order      delta-Y bit: 1 --> in proximity,   0 --> out of prox
** next           delta-Y bit: 1 --> touching tablet, 0 --> not touching
```

```
** high-order (sign) delta-Y bit: 1 --> first button closed, 0 --> open
**
**
** The lowest order bit of the delta-X bits is used to indicate that
** additional bytes follow: "1" indicates that the next byte is used for the
** next 7 barrel buttons with state.  The high order bit of each sequential
** byte in the series is "1" if an additional byte must be fetched, "0"
** otherwise.  In these additional bytes, the additional buttons are
** associated in order starting with the low-order bit.
**
**-----------------------------------------------------------------------*/



/**************************/
/* REFERENCE SECTION 27.0 */
/**************************/

/*-----------------------------------------------------------------------
** Skipped-point records:
** ---------------------
**
** (Taken from 8-bit byte delta X/Y: two bytes total)
**
**
**     |1|0| 0|0|0|0|1|0|   0|X|.|.|.|.|X|X|
**     -------------------------------------
**            (delta-X)          (delta-Y)
**
** An eight-bit delta with delta-X == 2, and delta-Y in the range
** (-4..3) indicates a count of elided points.  The delta-Y values in the
** range (-4..-1) are used to represent skip counts of (4..7).  If the
** delta-Y value is zero "0", the next two bytes are fetched to get a U16
** skip count value.
**
** The elided points are points removed between the point immediately prior
** to the skipped-point record and the point immediately afterward.  This
```

```
** implies that at least one point must follow every skip record (though
** the point may not appear next in the stream if there are intervening
** button state transitions). Reading applications that are interested in
** recovering elided points will typically interpolate. Skip counts of zero
** are meaningless and not permitted.
**
** Reserved:
** --------
**
** The remaining encodings from the 8-bit byte delta X/Y are reserved:
**
** delta-X of -4, -3, -2, -1, 3 AND ((delta-Y >= -4) & ((delta-Y <= 3))
**
**-----------------------------------------------------------------*/

#endif    // end of INKSTORE_INCLUDED
```